Serverless Architecture Project Structure

Function-as-a-Service architecture with TypeScript. AWS Lambda, API Gateway, and infrastructure as code.

#serverless #typescript #architecture #aws #lambda #sst

PNG PDF Copy Prompt

Project Directory

```
myproject/
    package.json
    tsconfig.json
    sst.config.ts  SST infrastruct…
    .gitignore
    .env.example
    README.md
  packages/  Monorepo packag…
    functions/  Lambda handlers
      package.json
      tsconfig.json
      src/
        api/  HTTP API handle…
          users/
            create.ts  POST /users
            get.ts  GET /users/:id
            list.ts  GET /users
          orders/
            create.ts
            get.ts
        events/  Event-triggered…
          user-created.ts  EventBridge han…
          order-placed.ts
        scheduled/  Cron jobs
          daily-report.ts
          cleanup.ts
        queues/  SQS consumers
          email-queue.ts
          notification-queue.ts
    core/  Shared business…
      package.json
      tsconfig.json
      src/
        index.ts  Public exports
        domain/
          user/
            user.ts  User entity
            user.service.ts
          order/
            order.ts
            order.service.ts
        repositories/
          user.repository.ts
          order.repository.ts
        events/
          publisher.ts  EventBridge pub…
          types.ts
    infra/  Infrastructure …
      api.ts  API Gateway rou…
      database.ts  DynamoDB tables
      events.ts  EventBridge rul…
      queues.ts  SQS queues
      storage.ts  S3 buckets
    tests/
      unit/
        services/
      integration/
        api/
```

Why This Structure?

This structure separates Lambda handlers (thin entry points) from shared business logic in `core/`. Each handler imports from core, keeping functions small. Infrastructure is defined as code in `infra/`, and the monorepo structure with SST enables type-safe resource binding.

Key Directories

packages/functions/ - Lambda handlers organized by trigger type (api, events, queues)

packages/core/ - Shared domain logic, services, and repositories

infra/ - SST/CDK stacks defining AWS resources

src/api/ - One file per endpoint: create.ts, get.ts, list.ts

Lambda Handler

```ts
// packages/functions/src/api/users/create.ts
import { UserService } from "@myproject/core";
import { ApiHandler } from "sst/node/api";

export const handler = ApiHandler(async (event) => {
  const body = JSON.parse(event.body ?? "{}");
  const user = await UserService.create(body);
  return { statusCode: 201, body: JSON.stringify(user) };
});
```

When To Use This

- Unpredictable or spiky traffic patterns
- Pay-per-use cost model preferred
- No server management overhead
- Event-driven workloads (S3, SQS, EventBridge)
- APIs with independent endpoint scaling

Lambda Triggers

API Gateway - HTTP endpoints → Lambda functions

EventBridge - Async events between services

SQS - Queue processing with retry and DLQ

Scheduled - Cron jobs via CloudWatch Events

Trade-offs

Cold starts - First invocation has latency, mitigate with provisioned concurrency

Execution limits - 15 min max runtime, 10GB memory, 6MB payload

Vendor lock-in - AWS-specific services, harder to migrate

Testing Strategy

Unit tests - Test core services with mocked repositories

Handler tests - Test Lambda handlers with event fixtures

Integration - Use SST's `sst dev` for live Lambda testing

Best Practices

- Keep handlers thin—delegate to core services
- Use environment variables for configuration
- Implement idempotency for event handlers
- Set up dead-letter queues for failed events
- Use structured logging with correlation IDs

Naming Conventions

Handlers - Match route: `users/create.ts` → POST /users

Stacks - By resource type: `api.ts`, `database.ts`

Events - Past tense: `user-created`, `order-placed`